Study guide: Finite difference methods for wave motion

Hans Petter Langtangen[1]

Center for Biomedical Computing, Simula Research Laboratory and Department of Informatics, University of Oslo[1]

Aug 28, 2014

Finite difference methods for waves on a string

Waves on a string can be modeled by the *wave equation*

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

$u(x,t)$ is the displacement of the string

Demo of waves on a string.

The complete initial-boundary value problem

$$\begin{aligned}
\frac{\partial^2 u}{\partial t^2} &= c^2 \frac{\partial^2 u}{\partial x^2}, & x \in (0,L),\ t \in (0,T] \quad (1)\\
u(x,0) &= I(x), & x \in [0,L] \quad (2)\\
\frac{\partial}{\partial t} u(x,0) &= 0, & x \in [0,L] \quad (3)\\
u(0,t) &= 0, & t \in (0,T] \quad (4)\\
u(L,t) &= 0, & t \in (0,T] \quad (5)
\end{aligned}$$

Input data in the problem

- Initial condition $u(x,0) = I(x)$: initial string shape
- Initial condition $u_t(x,0) = 0$: string starts from rest
- $c = \sqrt{T/\varrho}$: velocity of waves on the string
- (T is the tension in the string, ϱ is density of the string)
- Two boundary conditions on u: $u = 0$ means fixed ends (no displacement)

Rule for number of initial and boundary conditions:

- u_{tt} in the PDE: two initial conditions, on u and u_t
- u_t (and no u_{tt}) in the PDE: one initial conditions, on u
- u_{xx} in the PDE: one boundary condition on u at each boundary point

Demo of a vibrating string ($C = 0.8$)

- Our numerical method is sometimes exact (!)
- Our numerical method is sometimes subject to serious non-physical effects

Demo of a vibrating string ($C = 1.0012$)

Ooops!

Step 1: Discretizing the domain

Mesh in time:

$$0 = t_0 < t_1 < t_2 < \cdots < t_{N_t-1} < t_{N_t} = T \tag{6}$$

Mesh in space:

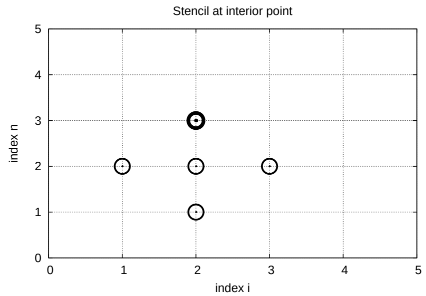
$$0 = x_0 < x_1 < x_2 < \cdots < x_{N_x-1} < x_{N_x} = L \tag{7}$$

Uniform mesh with constant mesh spacings Δt and Δx:

$$x_i = i\Delta x,\ i = 0,\ldots,N_x, \quad t_i = n\Delta t,\ n = 0,\ldots,N_t \tag{8}$$

The discrete solution

- The numerical solution is a mesh function: $u_i^n \approx u_e(x_i, t_n)$
- Finite difference stencil (or scheme): equation for u_i^n involving neighboring space-time points



Step 2: Fulfilling the equation at the mesh points

Let the PDE be satisfied at all *interior* mesh points:

$$\frac{\partial^2}{\partial t^2} u(x_i, t_n) = c^2 \frac{\partial^2}{\partial x^2} u(x_i, t_n), \tag{9}$$

for $i = 1,\ldots,N_x - 1$ and $n = 1,\ldots,N_t - 1$.

For $n = 0$ we have the initial conditions $u = I(x)$ and $u_t = 0$, and at the boundaries $i = 0, N_x$ we have the boundary condition $u = 0$.

Step 3: Replacing derivatives by finite differences

Widely used finite difference formula for the second-order derivative:

$$\frac{\partial^2}{\partial t^2} u(x_i, t_n) \approx \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = [D_t D_t u]_i^n$$

and

$$\frac{\partial^2}{\partial x^2} u(x_i, t_n) \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} = [D_x D_x u]_i^n$$

Step 3: Algebraic version of the PDE

Replace derivatives by differences:

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = c^2 \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}, \tag{10}$$

In operator notation:

$$[D_t D_t u = c^2 D_x D_x]_i^n \tag{11}$$

Step 3: Algebraic version of the initial conditions

- Need to replace the derivative in the initial condition $u_t(x, 0) = 0$ by a finite difference approximation
- The differences for u_{tt} and u_{xx} have second-order accuracy
- Use a centered difference for $u_t(x, 0)$

$$[D_{2t} u]_i^n = 0, \quad n = 0 \quad \Rightarrow \quad u_i^{n-1} = u_i^{n+1}, \quad i = 0,\ldots,N_x$$

The other initial condition $u(x, 0) = I(x)$ can be computed by

$$u_i^0 = I(x_i), \quad i = 0,\ldots,N_x$$

Step 4: Formulating a recursive algorithm

- Nature of the algorithm: compute u in space at $t = \Delta t, 2\Delta t, 3\Delta t, \ldots$
- Three time levels are involved in the general discrete equation: $n+1$, n, $n-1$
- u_i^n and u_i^{n-1} are then already computed for $i = 0, \ldots, N_x$, and u_i^{n+1} is the unknown quantity

Write out $[D_t D_t u = c^2 D_x D_x]_i^n$ and solve for u_i^{n+1},

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) \tag{12}$$

The Courant number

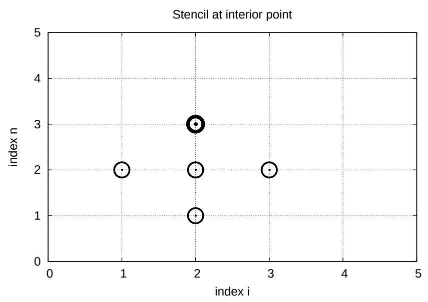
$$C = c\frac{\Delta t}{\Delta x}, \tag{13}$$

is known as the (dimensionless) *Courant number*

Observe.

There is only one parameter, C, in the discrete model: C lumps mesh parameters Δt and Δx with the only physical parameter, the wave velocity c. The value C and the smoothness of $I(x)$ govern the quality of the numerical solution.

The finite difference stencil



The stencil for the first time level

- Problem: the stencil for $n = 1$ involves u_i^{-1}, but time $t = -\Delta t$ is outside the mesh
- Remedy: use the initial condition $u_t = 0$ together with the stencil to eliminate u_i^{-1}

Initial condition:

$$[D_{2t} u = 0]_i^0 \quad \Rightarrow \quad u_i^{-1} = u_i^1$$

Insert in stencil $[D_t D_t u = c^2 D_x D_x]_i^0$ to get

$$u_i^1 = u_i^0 - \frac{1}{2}C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) \tag{14}$$

The algorithm

1. Compute $u_i^0 = I(x_i)$ for $i = 0, \ldots, N_x$
2. Compute u_i^1 by (14) and set $u_i^1 = 0$ for the boundary points $i = 0$ and $i = N_x$, for $n = 1, 2, \ldots, N - 1$,
3. For each time level $n = 1, 2, \ldots, N_t - 1$
   1. apply (12) to find u_i^{n+1} for $i = 1, \ldots, N_x - 1$
   2. set $u_i^{n+1} = 0$ for the boundary points $i = 0$, $i = N_x$.

Moving finite difference stencil

web page or a movie file.

Sketch of an implementation (1)

- Arrays:
  - u[i] stores u_i^{n+1}
  - u_1[i] stores u_i^n
  - u_2[i] stores u_i^{n-1}

Naming convention.

u is the unknown to be computed (a spatial mesh function), u_k is the computed spatial mesh function k time steps back in time.

PDE solvers should save memory

Important to minimize the memory usage.

The algorithm only needs to access the *three most recent time levels*, so we need only three arrays for u_i^{n+1}, u_i^n, and u_i^{n-1}, $i = 0, \ldots, N_x$. Storing all the solutions in a two-dimensional array of size $(N_x + 1) \times (N_t + 1)$ would be possible in this simple one-dimensional PDE problem, but not in large 2D problems and not even in small 3D problems.

Sketch of an implementation (2)

```
# Given mesh points as arrays x and t (x[i], t[n])
dx = x[1] - x[0]
dt = t[1] - t[0]
C = c*dt/dx            # Courant number
Nt = len(t)-1
C2 = C**2              # Help variable in the scheme

# Set initial condition u(x,0) = I(x)
for i in range(0, Nx+1):
    u_1[i] = I(x[i])

# Apply special formula for first step, incorporating du/dt=0
for i in range(1, Nx):
    u[i] = u_1[i] - 0.5*C**2(u_1[i+1] - 2*u_1[i] + u_1[i-1])
u[0] = 0;  u[Nx] = 0   # Enforce boundary conditions

# Switch variables before next step
u_2[:], u_1[:] = u_1, u

for n in range(1, Nt):
    # Update all inner mesh points at time t[n+1]
    for i in range(1, Nx):
        u[i] = 2u_1[i] - u_2[i] - \
               C**2(u_1[i+1] - 2*u_1[i] + u_1[i-1])

    # Insert boundary conditions
    u[0] = 0;  u[Nx] = 0
```

Verification

- Think about testing and verification before you start implementing the algorithm!
- Powerful testing tool: method of manufactured solutions and computation of convergence rates
- Will need a source term in the PDE and $u_t(x, 0) \neq 0$
- Even more powerful method: exact solution of the scheme

A slightly generalized model problem

Add source term f and nonzero initial condition $u_t(x, 0)$:

$$
\begin{aligned}
u_{tt} &= c^2 u_{xx} + f(x,t), & & (15)\\
u(x,0) &= I(x), & x \in [0, L] & \quad (16)\\
u_t(x,0) &= V(x), & x \in [0, L] & \quad (17)\\
u(0,t) &= 0, & t > 0, & \quad (18)\\
u(L,t) &= 0, & t > 0 & \quad (19)
\end{aligned}
$$

Discrete model for the generalized model problem

$$[D_t D_t u = c^2 D_x D_x + f]_i^n \tag{20}$$

Writing out and solving for the unknown u_i^{n+1}:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2(u_{i+1}^n - 2u_i^n + u_{i-1}^n) + \Delta t^2 f_i^n \tag{21}$$

Modified equation for the first time level

Centered difference for $u_t(x,0) = V(x)$:

$$[D_{2t}u = V]^0_i \quad \Rightarrow \quad u_i^{-1} = u_i^1 - 2\Delta t V_i,$$

Inserting this in the stencil (21) for $n = 0$ leads to

$$u_i^1 = u_i^0 - \Delta t V_i + \frac{1}{2}C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + \frac{1}{2}\Delta t^2 f_i^n \qquad (22)$$

Using an analytical solution of physical significance

- Standing waves occur in real life on a string
- Can be analyzed mathematically (known exact solution)

$$u_e(x,y,t)) = A\sin\left(\frac{\pi}{L}x\right)\cos\left(\frac{\pi}{L}ct\right) \qquad (23)$$

- PDE data: $f = 0$, boundary conditions $u_e(0,t) = u_e(L,0) = 0$, initial conditions $I(x) = A\sin\left(\frac{\pi}{L}x\right)$ and $V = 0$
- Note: $u_i^{n+1} \neq u_e(x_i, t_{n+1}$, and we do not know the error, so testing must aim at reproducing the expected convergence rates

Manufactured solution: principles

- Disadvantage with the previous physical solution: it does not test $V \neq 0$ and $f \neq 0$
- Method of manufactured solution:
  - Choose some $u_e(x,t)$
  - Insert in PDE and fit f
  - Set boundary and initial conditions compatible with the chosen $u_e(x,t)$

Manufactured solution: example

$$u_e(x,t) = x(L-x)\sin t$$

PDE $u_{tt} = c^2 u_{xx} + f$:

$$-x(L-x)\sin t = -2\sin t + f \quad \Rightarrow f = (2 - x(L-x))\sin t$$

Initial conditions become

$$u(x,0) = I(x) = 0$$
$$u_t(x,0) = V(x) = (2 - x(L-x))\cos t$$

Boundary conditions:

$$u(x,0) = u(x,L) = 0$$

Testing a manufactured solution

- Introduce common mesh parameter: $h = \Delta t$, $\Delta x = ch/C$
- This h keeps C and $\Delta t/\Delta x$ constant
- Select coarse mesh h: h_0
- Run experiments with $h_i = 2^{-i}h_0$ (halving the cell size), $i = 0,\ldots,m$
- Record the error E_i and h_i in each experiment
- Compute pariwise convergence rates $r_i = \ln E_{i+1}/E_i / \ln h_{i+1}/h_i$
- Verification: $r_i \rightarrow 2$ as i increases

Constructing an exact solution of the discrete equations

- Manufactured solution with computation of convergence rates: much manual work
- Simpler and more powerful: use an exact solution for u_i^n
- A linear or quadratic u_e in x and t is often a good candidate

Analytical work with the PDE problem

Here, choose u_e such that $u_e(x,0) = u_e(L,0) = 0$:

$$u_e(x,t) = x(L-x)(1+\frac{1}{2}t),$$

Insert in the PDE and find f:

$$f(x,t) = 2(1+t)c^2$$

Initial conditions:

$$I(x) = x(L-x), \quad V(x) = \frac{1}{2}x(L-x)$$

Analytical work with the discrete equations (1)

We want to show that u_e also solves the discrete equations!

Useful preliminary result:

$$[D_tD_tt^2]^n = \frac{t_{n+1}^2 - 2t_n^2 + t_{n-1}^2}{\Delta t^2} = (n+1)^2 - n^2 + (n-1)^2 = 2 \quad (24)$$

$$[D_tD_tt]^n = \frac{t_{n+1} - 2t_n + t_{n-1}}{\Delta t^2} = \frac{((n+1) - n + (n-1))\Delta t}{\Delta t^2} = 0 \quad (25)$$

Hence,

$$[D_tD_tu_e]_i^n = x_i(L-x_i)[D_tD_t(1+\frac{1}{2}t)]^n = x_i(L-x_i)\frac{1}{2}[D_tD_tt]^n = 0$$

Analytical work with the discrete equations (1)

$$[D_xD_xu_e]_i^n = (1+\frac{1}{2}t_n)[D_xD_x(xL-x^2)]_i = (1+\frac{1}{2}t_n)[LD_xD_xx - D_xD_xx^2$$

$$= -2(1+\frac{1}{2}t_n)$$

Now, $f_i^n = 2(1+\frac{1}{2}t_n)c^2$ and we get

$$[D_tD_tu_e - c^2D_xD_xu_e - f]_i^n = 0 - c^2(-1)2(1+\frac{1}{2}t_n + 2(1+\frac{1}{2}t_n)c^2 = 0$$

Moreover, $u_e(x_i,0) = I(x_i)$, $\partial u_e/\partial t = V(x_i)$ at $t=0$, and $u_e(x_0,t) = u_e(x_{N_x},0) = 0$. Also the modified scheme for the first time step is fulfilled by $u_e(x_i,t_n)$.

Testing with the exact discrete solution

- We have established that $u_i^{n+1} = u_e(x_i,t_{n+1}) = x_i(L-x_i)(1+t_{n+1}/2)$
- Run *one* simulation with one choice of c, Δt, and Δx
- Check that $\max_i |u_i^{n+1} - u_e(x_i,t_{n+1})| < \epsilon$, $\epsilon \sim 10^{-14}$ (machine precision + some round-off errors)
- This is the simplest and best verification test

Later we show that the exact solution of the discrete equations can be obtained by $C=1$ (!)

Implementation

The algorithm

1. Compute $u_i^0 = I(x_i)$ for $i = 0,\ldots,N_x$
2. Compute u_i^1 by (14) and set $u_i^1 = 0$ for the boundary points $i=0$ and $i=N_x$, for $n = 1,2,\ldots,N-1$,
3. For each time level $n = 1,2,\ldots,N_t-1$
   1. apply (12) to find u_i^{n+1} for $i = 1,\ldots,N_x-1$
   2. set $u_i^{n+1} = 0$ for the boundary points $i=0$, $i=N_x$.

What do to with the solution?

- Different problem settings demand different actions with the computed u_i^{n+1} at each time step
- Solution: let the solver function make a callback to a user function where the user can do whatever is desired with the solution
- Advantage: solver just solves and user uses the solution

```
def user_action(u, x, t, n):
    # u[i] at spatial mesh points x[i] at time t[n]
    # plot u
    # or store u
```

Making a solver function (1)

We specify Δt and C, and let the solver function compute $\Delta x = c\Delta t/C$.

```
def solver(I, V, f, c, L, dt, C, T, user_action=None):
    """Solve u_tt=c^2*u_xx + f on (0,L)x(0,T]."""
    Nt = int(round(T/dt))
    t = linspace(0, Nt*dt, Nt+1)   # Mesh points in time
    dx = dt*c/float(C)
    Nx = int(round(L/dx))
    x = linspace(0, L, Nx+1)       # Mesh points in space
    dx = x[1] - x[0]
    C2 = C**2                      # Help variable in the scheme
    if f is None or f == 0 :
        f = lambda x, t: 0
    if V is None or V == 0:
        V = lambda x: 0

    u   = zeros(Nx+1)   # Solution array at new time level
    u_1 = zeros(Nx+1)   # Solution at 1 time level back
    u_2 = zeros(Nx+1)   # Solution at 2 time levels back

    import time;  t0 = time.clock()  # for measuring CPU time

    # Load initial condition into u_1
    for i in range(0,Nx+1):
        u_1[i] = I(x[i])
```

Making a solver function (2)

```
def solver(I, V, f, c, L, dt, C, T, user_action=None):
    ...
    # Special formula for first time step
    n = 0
    for i in range(1, Nx):
        u[i] = u_1[i] + dt*V(x[i]) + \
               0.5*C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
               0.5*dt**2*f(x[i], t[n])
    u[0] = 0;  u[Nx] = 0

    if user_action is not None:
        user_action(u, x, t, 1)

    # Switch variables before next step
    u_2[:], u_1[:] = u_1, u
```

Making a solver function (3)

```
def solver(I, V, f, c, L, Nx, C, T, user_action=None):
    ...
    # Time loop

    for n in range(1, Nt):
        # Update all inner points at time t[n+1]
        for i in range(1, Nx):
            u[i] = - u_2[i] + 2*u_1[i] + \
                     C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
```

Verification: exact quadratic solution

Exact solution of the PDE problem *and* the discrete equations: $u_e(x,t) = x(L-x)(1+\frac{1}{2}t)$

```
import nose.tools as nt

def test_quadratic():
    """Check that u(x,t)=x(L-x)(1+t/2) is exactly reproduced."""
    def u_exact(x, t):
        return x*(L-x)*(1 + 0.5*t)

    def I(x):
        return u_exact(x, 0)

    def V(x):
        return 0.5*u_exact(x, 0)

    def f(x, t):
        return 2*(1 + 0.5*t)*c**2

    L = 2.5
    c = 1.5
    C = 0.75
    Nx = 3  # Very coarse mesh for this exact test
    dt = C*(L/Nx)/c
    T = 18

    u, x, t, cpu = solver(I, V, f, c, L, dt, C, T)
```

Visualization: animating $u(x,t)$

Make a viz function for animating the curve, with plotting in a user_action function plot_u:

```
def viz(I, V, f, c, L, dt, C, T, umin, umax, animate=True):
    """Run solver and visualize u at each time level."""
    import scitools.std as plt
    import time, glob, os

    def plot_u(u, x, t, n):
        """user_action function for solver."""
        plt.plot(x, u, 'r-',
                 xlabel='x', ylabel='u',
                 axis=[0, L, umin, umax],
                 title='t=%f' % t[n], show=True)
        # Let the initial condition stay on the screen for 2
        # seconds, else insert a pause of 0.2 s between each plot
        time.sleep(2) if t[n] == 0 else time.sleep(0.2)
        plt.savefig('frame_%04d.png' % n)  # for movie making

    # Clean up old movie frames
    for filename in glob.glob('frame_*.png'):
        os.remove(filename)

    user_action = plot_u if animate else None
    u, x, t, cpu = solver(I, V, f, c, L, dt, C, T, user_action)

    # Make movie files
```

Making movie files

- Store spatial curve in a file, for each time level
- Name files like 'something_%04d.png' % frame_counter
- Combine files to a movie

```
Terminal> scitools movie encoder=html output_file=movie.html \
          fps=4 frame_*.png  # web page with a player
Terminal> avconv -r 4 -i frame_%04d.png -c:v flv       movie.flv
Terminal> avconv -r 4 -i frame_%04d.png -c:v libtheora movie.ogg
Terminal> avconv -r 4 -i frame_%04d.png -c:v libx264   movie.mp4
Terminal> avconv -r 4 -i frame_%04d.png -c:v libpvx    movie.webm
```

Important.

- Zero padding (%04d) is essential for correct sequence of frames in something_*.png (Unix alphanumeric sort)
- Remove old frame_*.png files before making a new movie

Running a case

- Vibrations of a guitar string
- Triangular initial shape (at rest)

$$I(x) = \begin{cases} ax/x_0, & x < x_0 \\ a(L-x)/(L-x_0), & \text{otherwise} \end{cases} \quad (26)$$

Appropriate data:

- $L = 75$ cm, $x_0 = 0.8L$, $a = 5$ mm, time frequency $\nu = 440$ Hz

Implementation of the case

```python
def guitar(C):
    """Triangular wave (pulled guitar string)."""
    L = 0.75
    x0 = 0.8*L
    a = 0.005
    freq = 440
    wavelength = 2*L
    c = freq*wavelength
    omega = 2*pi*freq
    num_periods = 1
    T = 2*pi/omega*num_periods
    # Choose dt the same as the stability limit for Nx=50
    dt = L/50./c

    def I(x):
        return a*x/x0 if x < x0 else a/(L-x0)*(L-x)

    umin = -1.2*a;  umax = -umin
    cpu = viz(I, 0, 0, c, L, dt, C, T, umin, umax, animate=True)
```

Program: `wave1D_u0.py`.

Resulting movie for $C = 0.8$

Movie of the vibrating string

The benefits of scaling

- It is difficult to figure out all the physical parameters of a case
- And it is not necessary because of a powerful: *scaling*

Introduce new x, t, and u without dimension:

$$\bar{x} = \frac{x}{L}, \quad \bar{t} = \frac{c}{L}t, \quad \bar{u} = \frac{u}{a}$$

Insert this in the PDE (with $f = 0$) and dropping bars

$$u_{tt} = u_{xx}$$

Initial condition: set $a = 1$, $L = 1$, and $x_0 \in [0, 1]$ in (26).

In the code: set `a=c=L=1`, `x0=0.8`, and there is no need to calculate with wavelengths and frequencies to estimate c!

Just one challenge: determine the period of the waves and an appropriate end time (see the text for details).

Vectorization

- Problem: Python loops over long arrays are slow
- One remedy: use vectorized (`numpy`) code instead of explicit loops
- Other remedies: use Cython, port spatial loops to Fortran or C
- Speedup: 100-1000 (varies with N_x)

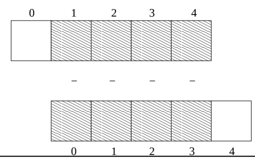
Next: vectorized loops

Operations on slices of arrays

- Introductory example: compute $d_i = u_{i+1} - u_i$

```python
n = u.size
for i in range(0, n-1):
    d[i] = u[i+1] - u[i]
```

- Note: all the differences here are independent of each other.
- Therefore $d = (u_1, u_2, \ldots, u_n) - (u_0, u_1, \ldots, u_{n-1})$
- In numpy code: `u[1:n] - u[0:n-1]` or just `u[1:] - u[:-1]`



Test the understanding

Newcomers to vectorization are encouraged to choose a small array u, say with five elements, and simulate with pen and paper both the loop version and the vectorized version.

Vectorization of finite difference schemes (1)

Finite difference schemes basically contains differences between array elements with shifted indices. Consider the updating formula

```
for i in range(1, n-1):
    u2[i] = u[i-1] - 2*u[i] + u[i+1]
```

The vectorization consists of replacing the loop by arithmetics on slices of arrays of length `n-2`:

```
u2 = u[:-2] - 2*u[1:-1] + u[2:]
u2 = u[0:n-2] - 2*u[1:n-1] + u[2:n]    # alternative
```

Note: `u2` gets length `n-2`.

If `u2` is already an array of length `n`, do update on "inner" elements

```
u2[1:-1]  = u[:-2] - 2*u[1:-1] + u[2:]
u2[1:n-1] = u[0:n-2] - 2*u[1:n-1] + u[2:n]    # alternative
```

Vectorization of finite difference schemes (2)

Include a function evaluation too:

```
def f(x):
    return x**2 + 1

# Scalar version
for i in range(1, n-1):
    u2[i] = u[i-1] - 2*u[i] + u[i+1] + f(x[i])

# Vectorized version
u2[1:-1] = u[:-2] - 2*u[1:-1] + u[2:] + f(x[1:-1])
```

Vectorized implementation in the solver function

Scalar loop:

```
for i in range(1, Nx):
    u[i] = 2*u_1[i] - u_2[i] + \
           C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])
```

Vectorized loop:

```
u[1:-1] = - u_2[1:-1] + 2*u_1[1:-1] + \
          C2*(u_1[:-2] - 2*u_1[1:-1] + u_1[2:])
```

or

```
u[1:Nx] = 2*u_1[1:Nx]- u_2[1:Nx] + \
          C2*(u_1[0:Nx-1] - 2*u_1[1:Nx] + u_1[2:Nx+1])
```

Program: `wave1D_u0v.py`

Verification of the vectorized version

```
def test_quadratic():
    """
    Check the scalar and vectorized versions work for
    a quadratic u(x,t)=x(L-x)(1+t/2) that is exactly reproduced.
    """
    # The following function must work for x as array or scalar
    u_exact = lambda x, t: x*(L - x)*(1 + 0.5*t)
    I = lambda x: u_exact(x, 0)
    V = lambda x: 0.5*u_exact(x, 0)
    # f is a scalar (zeros_like(x) works for scalar x too)
    f = lambda x, t: zeros_like(x) + 2*c**2*(1 + 0.5*t)

    L = 2.5
    c = 1.5
    C = 0.75
    Nx = 3  # Very coarse mesh for this exact test
    dt = C*(L/Nx)/c
    T = 18

    def assert_no_error(u, x, t, n):
        u_e = u_exact(x, t[n])
        diff = abs(u - u_e).max()
        nt.assert_almost_equal(diff, 0, places=13)

    solver(I, V, f, c, L, dt, C, T,
           user_action=assert_no_error, version='scalar')
    solver(I, V, f, c, L, dt, C, T,
           user_action=assert_no_error, version='vectorized')
```

Efficiency measurements

- Run `wave1D_u0v.py` for N_x as 50,100,200,400,800 and measuring the CPU time
- Observe substantial speed-up: vectorized version is about $N_x/5$ times faster

Much bigger improvements for 2D and 3D codes!

Generalization: reflecting boundaries

- Boundary condition $u=0$: u changes sign
- Boundary condition $u_x=0$: wave is perfectly reflected
- How can we implement u_x? (more complicated than $u=0$)

Demo of boundary conditions

Neumann boundary condition

$$\frac{\partial u}{\partial n} \equiv \mathbf{n}\cdot\nabla u = 0 \tag{27}$$

For a 1D domain $[0,L]$:

$$\left.\frac{\partial}{\partial n}\right|_{x=L} = \frac{\partial}{\partial x},\quad \left.\frac{\partial}{\partial n}\right|_{x=0} = -\frac{\partial}{\partial x}$$

Boundary condition terminology:

- u_x specified: Neumann condition
- u specified: Dirichlet condition

Discretization of derivatives at the boundary (1)

- How can we incorporate the condition $u_x=0$ in the finite difference scheme?
- We used centeral differences for u_{tt} and u_{xx}: $\mathcal{O}(\Delta t^2,\Delta x^2)$ accuracy
- Also for $u_t(x,0)$
- Should use central difference for u_x to preserve second order accuracy

$$\frac{u^n_{-1}-u^n_1}{2\Delta x} = 0 \tag{28}$$

Discretization of derivatives at the boundary (2)

$$\frac{u^n_{-1}-u^n_1}{2\Delta x} = 0$$

- Problem: u^n_{-1} is outside the mesh (fictitious value)
- Remedy: use the stencil at the boundary to eliminate u^n_{-1}; just replace u^n_{-1} by u^n_1

$$u^{n+1}_i = -u^{n-1}_i + 2u^n_i + 2C^2\left(u^n_{i+1}-u^n_i\right),\quad i=0 \tag{29}$$

Visualization of modified boundary stencil

Discrete equation for computing u^3_0 in terms of u^2_0, u^1_0, and u^2_1:

Animation in a web page or a movie file.

Implementation of Neumann conditions

- Use the general stencil for interior points also on the boundary
- Replace u^n_{i-1} by u^n_{i+1} for $i=0$
- Replace u^n_{i+1} by u^n_{i-1} for $i=N_x$

```
i = 0
ip1 = i+1
im1 = ip1  # i-1 -> i+1
u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])

i = Nx
im1 = i-1
ip1 = im1  # i+1 -> i-1
u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])

# Or just one loop over all points

for i in range(0, Nx+1):
    ip1 = i+1 if i < Nx else i-1
    im1 = i-1 if i > 0  else i+1
    u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])
```

Program `wave1D_dn0.py`

Moving finite difference stencil

web page or a movie file.

Index set notation

- Tedious to write index sets like $i = 0, \ldots, N_x$ and $n = 0, \ldots, N_t$
- Notation not valid if i or n starts at 1 instead...
- Both in math and code it is advantageous to use *index sets*
- $i \in \mathcal{I}_x$ instead of $i = 0, \ldots, N_x$
- Definition: $\mathcal{I}_x = \{0, \ldots, N_x\}$
- The first index: $i = \mathcal{I}_x^0$
- The last index: $i = \mathcal{I}_x^{-1}$
- All interior points: $i \in \mathcal{I}_x^i$, $\mathcal{I}_x^i = \{1, \ldots, N_x - 1\}$
- $\mathcal{I}_x^-$ means $\{0, \ldots, N_x - 1\}$
- $\mathcal{I}_x^+$ means $\{1, \ldots, N_x\}$

Index set notation in code

Notation	*Python*
$\mathcal{I}_x$	*Ix*
$\mathcal{I}_x^0$	*Ix[0]*
$\mathcal{I}_x^{-1}$	*Ix[-1]*
$\mathcal{I}_x^-$	*Ix[1:]*
$\mathcal{I}_x^+$	*Ix[:-1]*
$\mathcal{I}_x^i$	*Ix[1:-1]*

Index sets in action (1)

Index sets for a problem in the x, t plane:

$$\mathcal{I}_x = \{0, \ldots, N_x\}, \quad \mathcal{I}_t = \{0, \ldots, N_t\}, \tag{30}$$

Implemented in Python as

```
Ix = range(0, Nx+1)
It = range(0, Nt+1)
```

Index sets in action (2)

A finite difference scheme can with the index set notation be specified as

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right), \quad i \in \mathcal{I}_x^i,\ n \in \mathcal{I}_t^i$$
$$u_i = 0, \quad i = \mathcal{I}_x^0,\ n \in \mathcal{I}_t^i$$
$$u_i = 0, \quad i = \mathcal{I}_x^{-1},\ n \in \mathcal{I}_t^i$$

Corresponding implementation:

```
for n in It[1:-1]:
    for i in Ix[1:-1]:
        u[i] = - u_2[i] + 2*u_1[i] + \
               C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])
    i = Ix[0];  u[i] = 0
    i = Ix[-1]; u[i] = 0
```

Program `wave1D_dn.py`

Alternative implementation via ghost cells

- Instead of modifying the stencil at the boundary, we extend the mesh to cover u_{-1}^n and $u_{N_x+1}^n$
- The extra left and right cell are called *ghost cells*
- The extra points are called *ghost points*
- The u_{-1}^n and $u_{N_x+1}^n$ values are called *ghost values*
- Update ghost values as $u_{i-1}^n = u_{i+1}^n$ for $i = 0$ and $i = N_x$
- Then the stencil becomes right at the boundary

Implementation of ghost cells (1)

Add ghost points:

```
u   = zeros(Nx+3)
u_1 = zeros(Nx+3)
u_2 = zeros(Nx+3)

x = linspace(0, L, Nx+1)  # Mesh points without ghost points
```

- A major indexing problem arises with ghost cells since Python indices *must* start at 0.
- `u[-1]` will always mean the last element in `u`
- Math indexing: $-1, 0, 1, 2, \ldots, N_x + 1$
- Python indexing: `0,..,Nx+2`
- Remedy: use index sets

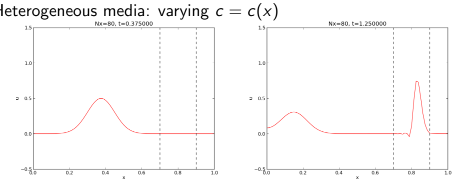
Implementation of ghost cells (2)

```
u = zeros(Nx+3)
Ix = range(1, u.shape[0]-1)

# Boundary values: u[Ix[0]], u[Ix[-1]]

# Set initial conditions
for i in Ix:
    u_1[i] = I(x[i-Ix[0]])  # Note i-Ix[0]

# Loop over all physical mesh points
for i in Ix:
    u[i] = - u_2[i] + 2*u_1[i] + \
           C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])

# Update ghost values
i = Ix[0]          # x=0 boundary
u[i-1] = u[i+1]
i = Ix[-1]         # x=L boundary
u[i-1] = u[i+1]
```

Program: `wave1D_dn0_ghost.py`.

Generalization: variable wave velocity

Heterogeneous media: varying $c = c(x)$



The model PDE with a variable coefficient

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) + f(x,t) \qquad (31)$$

This equation sampled at a mesh point (x_i, t_n):

$$\frac{\partial^2}{\partial t^2}u(x_i,t_n) = \frac{\partial}{\partial x}\left(q(x_i)\frac{\partial}{\partial x}u(x_i,t_n)\right) + f(x_i,t_n),$$

Discretizing the variable coefficient (1)

The principal idea is to *first discretize the outer derivative*.

Define

$$\phi = q(x)\frac{\partial u}{\partial x}$$

Then use a centered derivative around $x = x_i$ for the derivative of ϕ:

$$\left[\frac{\partial\phi}{\partial x}\right]_i^n \approx \frac{\phi_{i+\frac{1}{2}} - \phi_{i-\frac{1}{2}}}{\Delta x} = [D_x\phi]_i^n$$

Discretizing the variable coefficient (2)

Then discretize the inner operators:

$$\phi_{i+\frac{1}{2}} = q_{i+\frac{1}{2}}\left[\frac{\partial u}{\partial x}\right]_{i+\frac{1}{2}}^n \approx q_{i+\frac{1}{2}}\frac{u_{i+1}^n - u_i^n}{\Delta x} = [qD_x u]_{i+\frac{1}{2}}^n$$

Similarly,

$$\phi_{i-\frac{1}{2}} = q_{i-\frac{1}{2}}\left[\frac{\partial u}{\partial x}\right]_{i-\frac{1}{2}}^n \approx q_{i-\frac{1}{2}}\frac{u_i^n - u_{i-1}^n}{\Delta x} = [qD_x u]_{i-\frac{1}{2}}^n$$

Discretizing the variable coefficient (3)

These intermediate results are now combined to

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]^n_i \approx \frac{1}{\Delta x^2}\left(q_{i+\frac{1}{2}}\left(u^n_{i+1}-u^n_i\right) - q_{i-\frac{1}{2}}\left(u^n_i - u^n_{i-1}\right)\right) \tag{32}$$

In operator notation:

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]^n_i \approx [D_x q D_x u]^n_i \tag{33}$$

Remark.

Many are tempted to use the chain rule on the term $\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)$, but this is not a good idea!

Computing the coefficient between mesh points

- Given $q(x)$: compute $q_{i+\frac{1}{2}}$ as $q(x_{i+\frac{1}{2}})$
- Given q at the mesh points: q_i, use an average

$$q_{i+\frac{1}{2}} \approx \frac{1}{2}(q_i + q_{i+1}) = [\overline{q}^x]_i \quad \text{(arithmetic mean)} \tag{34}$$

$$q_{i+\frac{1}{2}} \approx 2\left(\frac{1}{q_i} + \frac{1}{q_{i+1}}\right)^{-1} \quad \text{(harmonic mean)} \tag{35}$$

$$q_{i+\frac{1}{2}} \approx (q_i q_{i+1})^{1/2} \quad \text{(geometric mean)} \tag{36}$$

The arithmetic mean in (34) is by far the most used averaging technique.

Discretization of variable-coefficient wave equation in operator notation

$$[D_t D_t u = D_x \overline{q}^x D_x u + f]^n_i \tag{37}$$

We clearly see the type of finite differences and averaging!

Write out and solve wrt u^{n+1}_i:

$$u^{n+1}_i = -u^{n-1}_i + 2u^n_i + \left(\frac{\Delta x}{\Delta t}\right)^2 \times \left(\frac{1}{2}(q_i + q_{i+1})(u^n_{i+1} - u^n_i) - \frac{1}{2}(q_i + q_{i-1})(u^n_i - u^n_{i-1})\right) + \Delta t^2 f^n_i \tag{38}$$

Neumann condition and a variable coefficient

Consider $\partial u/\partial x = 0$ at $x = L = N_x \Delta x$:

$$\frac{u^n_{i+1} - u^n_{i-1}}{2\Delta x} = 0 \quad u^n_{i+1} = u^n_{i-1}, \quad i = N_x$$

Insert $u^n_{i+1} = u^n_{i-1}$ in the stencil (38) for $i = N_x$ and obtain

$$u^{n+1}_i \approx -u^{n-1}_i + 2u^n_i + \left(\frac{\Delta x}{\Delta t}\right)^2 2q_i(u^n_{i-1} - u^n_i) + \Delta t^2 f^n_i$$

(We have used $q_{i+\frac{1}{2}} + q_{i-\frac{1}{2}} \approx 2q_i$.)

Alternative: assume $dq/dx = 0$ (simpler).

Implementation of variable coefficients

Assume `c[i]` holds c_i the spatial mesh points

```
for i in range(1, Nx):
    u[i] = - u_2[i] + 2*u_1[i] + \
           C2*(0.5*(q[i] + q[i+1])*(u_1[i+1] - u_1[i])  - \
               0.5*(q[i] + q[i-1])*(u_1[i] - u_1[i-1])) + \
           dt2*f(x[i], t[n])
```

Here: `C2=(dt/dx)**2`

Vectorized version:

```
u[1:-1] = - u_2[1:-1] + 2*u_1[1:-1] + \
          C2*(0.5*(q[1:-1] + q[2:])*(u_1[2:] - u_1[1:-1]) -
              0.5*(q[1:-1] + q[:-2])*(u_1[1:-1] - u_1[:-2])) + \
          dt2*f(x[1:-1], t[n])
```

Neumann condition $u_x = 0$: same ideas as in 1D (modified stencil or ghost cells).

A more general model PDE with variable coefficients

$$\varrho(x)\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) + f(x,t) \tag{39}$$

A natural scheme is

$$[\varrho D_t D_t u = D_x \overline{q}^x D_x u + f]^n_i \tag{40}$$

Or

$$[D_t D_t u = \varrho^{-1} D_x \overline{q}^x D_x u + f]^n_i \tag{41}$$

No need to average ϱ, just sample at i

Generalization: damping

Why do waves die out?

- Damping (non-elastic effects, air resistance)
- 2D/3D: conservation of energy makes an amplitude reduction by $1/\sqrt{r}$ (2D) or $1/r$ (3D)

Simplest damping model (for physical behavior, see demo):

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = c^2\frac{\partial^2 u}{\partial x^2} + f(x,t), \tag{42}$$

$b \geq 0$: prescribed damping coefficient.

Discretization via centered differences to ensure $\mathcal{O}(\Delta t^2)$ error:

$$[D_tD_tu + bD_{2t}u = c^2D_xD_xu + f]_i^n \tag{43}$$

Need special formula for u_i^1 + special stencil (or ghost cells) for Neumann conditions.

Building a general 1D wave equation solver

The program `wave1D_dn_vc.py` solves a fairly general 1D wave equation:

$$\begin{aligned}
u_t &= (c^2(x)u_x)_x + f(x,t), && x\in(0,L),\ t\in(0,T] && (44)\\
u(x,0) &= I(x), && x\in[0,L] && (45)\\
u_t(x,0) &= V(t), && x\in[0,L] && (46)\\
u(0,t) &= U_0(t) \text{ or } u_x(0,t)=0, && t\in(0,T] && (47)\\
u(L,t) &= U_L(t) \text{ or } u_x(L,t)=0, && t\in(0,T] && (48)
\end{aligned}$$

Can be adapted to many needs.

Collection of initial conditions

The function `pulse` in `wave1D_dn_vc.py` offers four initial conditions:

1. a rectangular pulse ("plug")
2. a Gaussian function (`gaussian`)
3. a "cosine hat": one period of $1+\cos(\pi x, x\in[-1,1]$
4. half a "cosine hat": half a period of $\cos\pi x$, $x\in[-\frac{1}{2},\frac{1}{2}]$

Can locate the initial pulse at $x=0$ or in the middle

```
>>> import wave1D_dn_vc as w
>>> w.pulse(loc='left', pulse_tp='cosinehat', Nx=50, every_frame=10
```

Finite difference methods for 2D and 3D wave equations

Constant wave velocity c:

$$\frac{\partial^2 u}{\partial t^2} = c^2\nabla^2 u \text{ for } \boldsymbol{x}\in\Omega\subset\mathbb{R}^d,\ t\in(0,T] \tag{49}$$

Variable wave velocity:

$$\varrho\frac{\partial^2 u}{\partial t^2} = \nabla\cdot(q\nabla u) + f \text{ for } \boldsymbol{x}\in\Omega\subset\mathbb{R}^d,\ t\in(0,T] \tag{50}$$

Examples on wave equations written out in 2D/3D

3D, constant c:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$$

2D, variable c:

$$\varrho(x,y)\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + f(x,y,t) \tag{51}$$

Compact notation:

$$\begin{aligned}
u_{tt} &= c^2(u_{xx}+u_{yy}+u_{zz}) + f, && (52)\\
\varrho u_{tt} &= (qu_x)_x + (qu_z)_z + (qu_z)_z + f && (53)
\end{aligned}$$

Boundary and initial conditions

We need *one* boundary condition at *each point* on $\partial\Omega$:

1. u is prescribed ($u=0$ or known incoming wave)
2. $\partial u/\partial n = \boldsymbol{n}\cdot\nabla u$ prescribed ($=0$: reflecting boundary)
3. open boundary (radiation) condition: $u_t + \boldsymbol{c}\cdot\nabla u = 0$ (let waves travel undisturbed out of the domain)

PDEs with *second-order* time derivative need *two* initial conditions:

1. $u = I$,
2. $u_t = V$.

Mesh

- Mesh point: (x_i, y_j, z_k, t_n)
- x direction: $x_0 < x_1 < \cdots < x_{N_x}$
- y direction: $y_0 < y_1 < \cdots < y_{N_y}$
- z direction: $z_0 < z_1 < \cdots < z_{N_z}$
- $u^n_{i,j,k} \approx u_e(x_i, y_j, z_k, t_n)$

Discretization

$$[D_tD_t u = c^2(D_xD_x u + D_yD_y u) + f]^n_{i,j,k},$$

Written out in detail:

$$\frac{u^{n+1}_{i,j} - 2u^n_{i,j} + u^{n-1}_{i,j}}{\Delta t^2} = c^2\frac{u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}}{\Delta x^2} + c^2\frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}}{\Delta y^2} + f^n_{i,j},$$

$u^{n-1}_{i,j}$ and $u^n_{i,j}$ are known, solve for $u^{n+1}_{i,j}$:

$$u^{n+1}_{i,j} = 2u^n_{i,j} + u^{n-1}_{i,j} + c^2\Delta t^2[D_xD_x u + D_yD_y u]^n_{i,j}$$

Special stencil for the first time step

- The stencil for $u^1_{i,j}$ $(n=0)$ involves $u^{-1}_{i,j}$ which is outside the time mesh
- Remedy: use discretized $u_t(x,0) = V$ and the stencil for $n=0$ to develop a special stencil (as in the 1D case)

$$[D_{2t} u = V]^0_{i,j} \quad \Rightarrow \quad u^{-1}_{i,j} = u^1_{i,j} - 2\Delta t V_{i,j}$$

$$u^{n+1}_{i,j} = u^n_{i,j} - 2\Delta V_{i,j} + \frac{1}{2}c^2\Delta t^2[D_xD_x u + D_yD_y u]^n_{i,j}$$

Variable coefficients (1)

3D wave equation:

$$\varrho u_{tt} = (qu_x)_x + (qu_y)_y + (qu_z)_z + f(x,y,z,t)$$

Just apply the 1D discretization for each term:

$$[\varrho D_tD_t u = (D_x\overline{q}^x D_x u + D_y\overline{q}^y D_y u + D_z\overline{q}^z D_z u) + f]^n_{i,j,k} \qquad (54)$$

Need special formula for $u^1_{i,j,k}$ (use $[D_{2t}u = V]^0$ and stencil for $n=0$).

Variable coefficients (2)

Written out:

$$\begin{aligned}
u^{n+1}_{i,j,k} &= -u^{n-1}_{i,j,k} + 2u^n_{i,j,k} + \\
&= \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta x^2}(\frac{1}{2}(q_{i,j,k} + q_{i+1,j,k})(u^n_{i+1,j,k} - u^n_{i,j,k}) - \\
&\qquad \frac{1}{2}(q_{i-1,j,k} + q_{i,j,k})(u^n_{i,j,k} - u^n_{i-1,j,k})) + \\
&= \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta x^2}(\frac{1}{2}(q_{i,j,k} + q_{i,j+1,k})(u^n_{i,j+1,k} - u^n_{i,j,k}) - \\
&\qquad \frac{1}{2}(q_{i,j-1,k} + q_{i,j,k})(u^n_{i,j,k} - u^n_{i,j-1,k})) + \\
&= \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta x^2}(\frac{1}{2}(q_{i,j,k} + q_{i,j,k+1})(u^n_{i,j,k+1} - u^n_{i,j,k}) - \\
&\qquad \frac{1}{2}(q_{i,j,k-1} + q_{i,j,k})(u^n_{i,j,k} - u^n_{i,j,k-1})) + \\
+ &\quad \Delta t^2 f^n_{i,j,k}
\end{aligned}$$

Neumann boundary condition in 2D

Use ideas from 1D! Example: $\frac{\partial u}{\partial n}$ at $y=0$, $\frac{\partial u}{\partial n} = -\frac{\partial u}{\partial y}$

Boundary condition discretization:

$$[-D_{2y} u = 0]^n_{i,0} \quad \Rightarrow \quad \frac{u^n_{i,1} - u^n_{i,-1}}{2\Delta y} = 0,\ i \in \mathcal{I}_x$$

Insert $u^n_{i,-1} = u^n_{i,1}$ in the stencil for $u^{n+1}_{i,j=0}$ to obtain a modified stencil on the boundary.

Pattern: use interior stencil also on the bundary, but replace $j-1$ by $j+1$

Alternative: use ghost cells and ghost values

Implementation of 2D/3D problems

$$u_t = c^2(u_{xx} + u_{yy}) + f(x, y, t), \quad (x, y) \in \Omega,\ t \in (0, T] \tag{55}$$

$$u(x, y, 0) = I(x, y), \quad (x, y) \in \Omega \tag{56}$$

$$u_t(x, y, 0) = V(x, y), \quad (x, y) \in \Omega \tag{57}$$

$$u = 0, \quad (x, y) \in \partial\Omega,\ t \in (0, T] \tag{58}$$

$\Omega = [0, L_x] \times [0, L_y]$

Discretization:

$$[D_tD_t u = c^2(D_xD_x u + D_yD_y u) + f]^n_{i,j},$$

Algorithm

1. Set initial condition $u^0_{i,j} = I(x_i, y_j)$
2. Compute $u^1_{i,j} = \cdots$ for $i \in \mathcal{I}^i_x$ and $j \in \mathcal{I}^i_y$
3. Set $u^1_{i,j} = 0$ for the boundaries $i = 0, N_x$, $j = 0, N_y$
4. For $n = 1, 2, \ldots, N_t$:
    1. Find $u^{n+1}_{i,j} = \cdots$ for $i \in \mathcal{I}^i_x$ and $j \in \mathcal{I}^i_y$
    2. Set $u^{n+1}_{i,j} = 0$ for the boundaries $i = 0, N_x$, $j = 0, N_y$

Scalar computations: mesh

Program: `wave2D_u0.py`

```
def solver(I, V, f, c, Lx, Ly, Nx, Ny, dt, T,
           user_action=None, version='scalar'):
```

Mesh:

```
x = linspace(0, Lx, Nx+1)                  # mesh points in x dir
y = linspace(0, Ly, Ny+1)                  # mesh points in y dir
dx = x[1] - x[0]
dy = y[1] - y[0]
Nt = int(round(T/float(dt)))
t = linspace(0, N*dt, N+1)                 # mesh points in time
Cx2 = (c*dt/dx)**2;  Cy2 = (c*dt/dy)**2    # help variables
dt2 = dt**2
```

Scalar computations: arrays

Store $u^{n+1}_{i,j}$, $u^n_{i,j}$, and $u^{n-1}_{i,j}$ in three two-dimensional arrays:

```
u   = zeros((Nx+1,Ny+1))    # solution array
u_1 = zeros((Nx+1,Ny+1))    # solution at t-dt
u_2 = zeros((Nx+1,Ny+1))    # solution at t-2*dt
```

$u^{n+1}_{i,j}$ corresponds to `u[i,j]`, etc.

Scalar computations: initial condition

```
Ix = range(0, u.shape[0])
Iy = range(0, u.shape[1])
It = range(0, t.shape[0])

for i in Ix:
    for j in Iy:
        u_1[i,j] = I(x[i], y[j])

if user_action is not None:
    user_action(u_1, x, xv, y, yv, t, 0)
```

Arguments xv and yv: for vectorized computations

Scalar computations: primary stencil

```
def advance_scalar(u, u_1, u_2, f, x, y, t, n, Cx2, Cy2, dt2,
                   V=None, step1=False):
    Ix = range(0, u.shape[0]);  Iy = range(0, u.shape[1])
    if step1:
        dt = sqrt(dt2)  # save
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;  D2 = 0
    else:
        D1 = 2;  D2 = 1
    for i in Ix[1:-1]:
        for j in Iy[1:-1]:
            u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
            u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
            u[i,j] = D1*u_1[i,j] - D2*u_2[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f(x[i], y[j], t[n])
            if step1:
                u[i,j] += dt*V(x[i], y[j])
    # Boundary condition u=0
    j = Iy[0]
    for i in Ix: u[i,j] = 0
    j = Iy[-1]
    for i in Ix: u[i,j] = 0
    i = Ix[0]
    for j in Iy: u[i,j] = 0
    i = Ix[-1]
    for j in Iy: u[i,j] = 0
    return u
```

Vectorized computations: mesh coordinates

Mesh with 30×30 cells: vectorization reduces the CPU time by a factor of 70 (!).

Need special coordinate arrays `xv` and `yv` such that $I(x, y)$ and $f(x, y, t)$ can be vectorized:

```
from numpy import newaxis
xv = x[:,newaxis]
yv = y[newaxis,:]

u_1[:,:] = I(xv, yv)
f_a[:,:] = f(xv, yv, t)
```

Vectorized computations: stencil

```
def advance_vectorized(u, u_1, u_2, f_a, Cx2, Cy2, dt2,
                       V=None, step1=False):
    if step1:
        dt = sqrt(dt2)  # save
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;  D2 = 0
    else:
        D1 = 2;  D2 = 1
    u_xx = u_1[:-2,1:-1] - 2*u_1[1:-1,1:-1] + u_1[2:,1:-1]
    u_yy = u_1[1:-1,:-2] - 2*u_1[1:-1,1:-1] + u_1[1:-1,2:]
    u[1:-1,1:-1] = D1*u_1[1:-1,1:-1] - D2*u_2[1:-1,1:-1] + \
                   Cx2*u_xx + Cy2*u_yy + dt2*f_a[1:-1,1:-1]
    if step1:
        u[1:-1,1:-1] += dt*V[1:-1, 1:-1]
    # Boundary condition u=0
    j = 0
    u[:,j] = 0
    j = u.shape[1]-1
    u[:,j] = 0
    i = 0
    u[i,:] = 0
    i = u.shape[0]-1
    u[i,:] = 0
    return u
```

Verification: quadratic solution (1)

Manufactured solution:

$$u_e(x, y, t) = x(L_x - x)y(L_y - y)(1 + \frac{1}{2}t) \quad (59)$$

Requires $f = 2c^2(1 + \frac{1}{2}t)(y(L_y - y) + x(L_x - x))$.

This u_e is ideal because it also solves the discrete equations!

Verification: quadratic solution (2)

- $[D_tD_t1]^n = 0$
- $[D_tD_tt]^n = 0$
- $[D_tD_tt^2] = 2$
- D_tD_t is a linear operator:
  $[D_tD_t(au + bv)]^n = a[D_tD_tu]^n + b[D_tD_tv]^n$

$$\begin{aligned}[D_xD_xu_e]^n_{i,j} &= [y(L_y - y)(1 + \frac{1}{2}t)D_xD_xx(L_x - x)]^n_{i,j}\\ &= y_j(L_y - y_j)(1 + \frac{1}{2}t_n)2\end{aligned}$$

- Similar calculations for $[D_yD_yu_e]^n_{i,j}$ and $[D_tD_tu_e]^n_{i,j}$ terms
- Must also check the equation for $u^1_{i,j}$

Migrating loops to Cython

- Vectorization: 5-10 times slower than pure C or Fortran code
- Cython: extension of Python for translating functions to C
- Principle: declare variables with type

Declaring variables and annotating the code

Pure Python code:

```
def advance_scalar(u, u_1, u_2, f, x, y, t,
                   n, Cx2, Cy2, dt2, D1=2, D2=1):
    Ix = range(0, u.shape[0]);  Iy = range(0, u.shape[1])
    for i in Ix[1:-1]:
        for j in Iy[1:-1]:
            u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
            u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
            u[i,j] = D1*u_1[i,j] - D2*u_2[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f(x[i], y[j], t[n])
```

- Copy this function and put it in a file with `.pyx` extension.
- Add type of variables:
  - `function(a, b)` → `cpdef function(int a, double b)`
  - `v = 1.2` → `cdef double v = 1.2`
  - Array declaration:
    `np.ndarray[np.float64_t, ndim=2, mode='c'] u`

Cython version of the functions

```
import numpy as np
cimport numpy as np
cimport cython
ctypedef np.float64_t DT    # data type

@cython.boundscheck(False)  # turn off array bounds check
@cython.wraparound(False)   # turn off negative indices (u[-1,-1])
cpdef advance(
    np.ndarray[DT, ndim=2, mode='c'] u,
    np.ndarray[DT, ndim=2, mode='c'] u_1,
    np.ndarray[DT, ndim=2, mode='c'] u_2,
    np.ndarray[DT, ndim=2, mode='c'] f,
    double Cx2, double Cy2, double dt2):

    cdef int Nx, Ny, i, j
    cdef double u_xx, u_yy
    Nx = u.shape[0]-1
    Ny = u.shape[1]-1
    for i in xrange(1, Nx):
        for j in xrange(1, Ny):
            u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
            u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
            u[i,j] = 2*u_1[i,j] - u_2[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f[i,j]
```

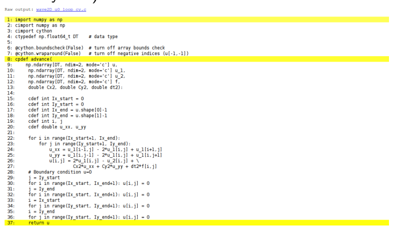
Note: from now in we skip the code for setting boundary values

Visual inspection of the C translation

See how effective Cython can translate this code to C:

```
Terminal> cython -a wave2D_u0_loop_cy.pyx
```

Load `wave2D_u0_loop_cy.html` in a browser (white: pure C, yellow: still Python):



Can click on `wave2D_u0_loop_cy.c` to see the generated C code...

Building the extension module

- Cython code must be translated to C
- C code must be compiled
- Compiled C code must be linked to Python C libraries
- Result: *C extension module* (`.so` file) that can be loaded as a standard Python module
- Use a `setup.py` script to build the extension module

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

cymodule = 'wave2D_u0_loop_cy'
setup(
  name=cymodule
  ext_modules=[Extension(cymodule, [cymodule + '.pyx'],)],
  cmdclass={'build_ext': build_ext},
)

Terminal> python setup.py build_ext --inplace
```

Calling the Cython function from Python

```
import wave2D_u0_loop_cy
advance = wave2D_u0_loop_cy.advance
...
for n in It[1:-1]:                      # time loop
    f_a[:,:] = f(xv, yv, t[n])      # precompute, size as u
    u = advance(u, u_1, u_2, f_a, x, y, t, Cx2, Cy2, dt2)
```

Efficiency:

- 120×120 cells in space:
  - Pure Python: 1370 CPU time units
  - Vectorized numpy: 5.5
  - Cython: 1
- 60×60 cells in space:
  - Pure Python: 1000 CPU time units
  - Vectorized numpy: 6
  - Cython: 1

Migrating loops to Fortran

- Write the advance function in pure Fortran
- Use `f2py` to generate C code for calling Fortran from Python
- Full manual control of the translation to Fortran

The Fortran subroutine

```
      subroutine advance(u, u_1, u_2, f, Cx2, Cy2, dt2, Nx, Ny)
      integer Nx, Ny
      real*8 u(0:Nx,0:Ny), u_1(0:Nx,0:Ny), u_2(0:Nx,0:Ny)
      real*8 f(0:Nx, 0:Ny), Cx2, Cy2, dt2
      integer i, j
Cf2py intent(in, out) u

C     Scheme at interior points
      do j = 1, Ny-1
         do i = 1, Nx-1
            u(i,j) = 2*u_1(i,j) - u_2(i,j) +
     &      Cx2*(u_1(i-1,j) - 2*u_1(i,j) + u_1(i+1,j)) +
     &      Cy2*(u_1(i,j-1) - 2*u_1(i,j) + u_1(i,j+1)) +
     &      dt2*f(i,j)
         end do
      end do
```

Note: `Cf2py` comment declares `u` as input argument and return value back to Python

Building the Fortran module with f2py

```
Terminal> f2py -m wave2D_u0_loop_f77 -h wave2D_u0_loop_f77.pyf \
          --overwrite-signature wave2D_u0_loop_f77.f
Terminal> f2py -c wave2D_u0_loop_f77.pyf --build-dir build_f77 \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_f77.f
```

f2py changes the argument list (!)

```
>>> import wave2D_u0_loop_f77
>>> print wave2D_u0_loop_f77.__doc__
This module 'wave2D_u0_loop_f77' is auto-generated with f2py....
Functions:
  u = advance(u,u_1,u_2,f,cx2,cy2,dt2,
      nx=(shape(u,0)-1),ny=(shape(u,1)-1))
```

- Array limits have default values
- Examine doc strings from f2py!

How to avoid array copying

- Two-dimensional arrays are stored row by row in Python and C
- Two-dimensional arrays are stored column by column in Fortran
- f2py takes a copy of a numpy (C) array and transposes it when calling Fortran
- Such copies are time and memory consuming
- Remedy: declare numpy arrays with Fortran storage

```
order = 'Fortran' if version == 'f77' else 'C'
u   = zeros((Nx+1,Ny+1), order=order)
u_1 = zeros((Nx+1,Ny+1), order=order)
u_2 = zeros((Nx+1,Ny+1), order=order)
```

Option -DF2PY_REPORT_ON_ARRAY_COPY=1 makes f2py write out array copying:

```
Terminal> f2py -c wave2D_u0_loop_f77.pyf --build-dir build_f77 \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_f77.f
```

Efficiency of translating to Fortran

- Same efficiency (in this example) as Cython and C
- About 5 times faster than vectorized numpy code
- $>$ 1000 faster than pure Python code

Migrating loops to C via Cython

- Write the advance function in pure C
- Use Cython to generate C code for calling C from Python
- Full manual control of the translation to C

The C code

- numpy arrays transferred to C are one-dimensional in C
- Need to translate [i,j] indices to single indices

```c
/* Translate (i,j) index to single array index */
#define idx(i,j) (i)*(Ny+1) + j

void advance(double* u, double* u_1, double* u_2, double* f,
     double Cx2, double Cy2, double dt2,
     int Nx, int Ny)
{
  int i, j;
  /* Scheme at interior points */
  for (i=1; i<=Nx-1; i++) {
    for (j=1; j<=Ny-1; j++) {
        u[idx(i,j)] = 2*u_1[idx(i,j)] - u_2[idx(i,j)] +
        Cx2*(u_1[idx(i-1,j)] - 2*u_1[idx(i,j)] + u_1[idx(i+1,j)]) +
        Cy2*(u_1[idx(i,j-1)] - 2*u_1[idx(i,j)] + u_1[idx(i,j+1)]) +
        dt2*f[idx(i,j)];
}
    }
  }
}
```

The Cython interface file

```python
import numpy as np
cimport numpy as np
cimport cython

cdef extern from "wave2D_u0_loop_c.h":
    void advance(double* u, double* u_1, double* u_2, double* f,
                 double Cx2, double Cy2, double dt2,
                 int Nx, int Ny)

@cython.boundscheck(False)
@cython.wraparound(False)
def advance_cwrap(
    np.ndarray[double, ndim=2, mode='c'] u,
    np.ndarray[double, ndim=2, mode='c'] u_1,
    np.ndarray[double, ndim=2, mode='c'] u_2,
    np.ndarray[double, ndim=2, mode='c'] f,
    double Cx2, double Cy2, double dt2):
    advance(&u[0,0], &u_1[0,0], &u_2[0,0], &f[0,0],
            Cx2, Cy2, dt2,
            u.shape[0]-1, u.shape[1]-1)
    return u
```

Building the extension module

Compile and link the extension module with a `setup.py` file:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

sources = ['wave2D_u0_loop_c.c', 'wave2D_u0_loop_c_cy.pyx']
module = 'wave2D_u0_loop_c_cy'
setup(
  name=module,
  ext_modules=[Extension(module, sources,
                         libraries=[], # C libs to link with
                         )],
  cmdclass={'build_ext': build_ext},
)
```

```
Terminal> python setup.py build_ext --inplace
```

In Python:

```
import wave2D_u0_loop_c_cy
advance = wave2D_u0_loop_c_cy.advance_cwrap
...
f_a[:,:] = f(xv, yv, t[n])
u = advance(u, u_1, u_2, f_a, Cx2, Cy2, dt2)
```

Migrating loops to C via f2py

- Write the advance function in pure C
- Use `f2py` to generate C code for calling C from Python
- Full manual control of the translation to C

The C code and the Fortran interface file

- Write the C function `advance` as before
- Write a Fortran 90 module defining the signature of the `advance` function
- Or: write a Fortran 77 function defining the signature and let `f2py` generate the Fortran 90 module

Fortran 77 signature (note `intent(c)`):

```
      subroutine advance(u, u_1, u_2, f, Cx2, Cy2, dt2, Nx, Ny)
Cf2py intent(c) advance
      integer Nx, Ny, N
      real*8 u(0:Nx,0:Ny), u_1(0:Nx,0:Ny), u_2(0:Nx,0:Ny)
      real*8 f(0:Nx, 0:Ny), Cx2, Cy2, dt2
Cf2py intent(in, out) u
Cf2py intent(c) u, u_1, u_2, f, Cx2, Cy2, dt2, Nx, Ny
      return
      end
```

Building the extension module

Generate Fortran 90 module (`wave2D_u0_loop_c_f2py.pyf`):

```
Terminal> f2py -m wave2D_u0_loop_c_f2py \
          -h wave2D_u0_loop_c_f2py.pyf --overwrite-signature \
          wave2D_u0_loop_c_f2py_signature.f
```

The compile and build step must list the C files:

```
Terminal> f2py -c wave2D_u0_loop_c_f2py.pyf \
          --build-dir tmp_build_c \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_c.c
```

Migrating loops to C++ via f2py

- C++ can be used as an alternative to C
- C++ code often applies sophisticated arrays
- Challenge: translate from `numpy` C arrays to C++ array classes
- Can use SWIG to make C++ classes available as Python classes
- Easier (and more efficient):
  - Make C API to the C++ code
  - Wrap C API with `f2py`
  - Send numpy arrays to C API and let C translate numpy arrays into C++ array classes

Analysis of the difference equations

Properties of the solution of the wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

Solutions:

$$u(x,t) = g_R(x - ct) + g_L(x + ct), \tag{60}$$

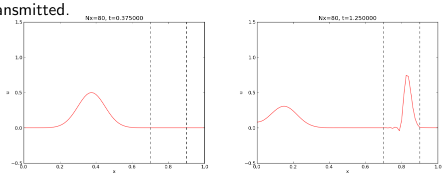
If $u(x,0) = I(x)$ and $u_t(x,0) = 0$:

$$u(x,t) = \frac{1}{2} I(x - ct) + \frac{1}{2} I(x + ct) \tag{61}$$

Two waves: one traveling to the right and one to the left

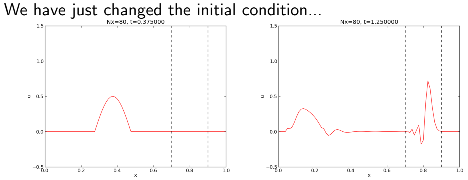
Effect of variable wave velocity

A wave propagates perfectly ($C = 1$) and hits a medium with $1/4$ of the wave velocity. A part of the wave is reflected and the rest is transmitted.



What happens here?

We have just changed the initial condition...



Representation of waves as sum of sine/cosine waves

Build $I(x)$ of wave components $e^{ikx} = \cos kx + i \sin kx$:

$$I(x) \approx \sum_{k \in K} b_k e^{ikx} \tag{62}$$

- k is the frequency of a component ($\lambda = 2\pi/k$ corresponding wave length)
- K is some set of all k needed to approximate $I(x)$ well
- b_k must be computed (Fourier coefficients)

Since $u(x,t) = \frac{1}{2} I(x - ct) + \frac{1}{2} I(x + ct)$:

$$u(x,t) = \frac{1}{2} \sum_{k \in K} b_k e^{ik(x-ct)} + \frac{1}{2} \sum_{k \in K} b_k e^{ik(x+ct)} \tag{63}$$

Our interest: one component $e^{i(kx - \omega t)}$, $\omega = kc$

Analysis of the finite difference scheme

A similar discrete $u_q^n = e^{i(kx_q - \tilde{\omega} t_n)}$ solves

$$[D_t D_t u = c^2 D_x D_x u]_q^n \tag{64}$$

Note: different frequency $\tilde{\omega} \neq \omega$

- How accurate is $\tilde{\omega}$ compared to ω?
- What about the wave amplitude?

Preliminary results

$$[D_t D_t e^{i\omega t}]^n = -\frac{4}{\Delta t^2} \sin^2\left(\frac{\omega \Delta t}{2}\right) e^{i\omega n \Delta t}$$

By $\omega \rightarrow k$, $t \rightarrow x$, $n \rightarrow q$) it follows that

$$[D_x D_x e^{ikx}]_q = -\frac{4}{\Delta x^2} \sin^2\left(\frac{k \Delta x}{2}\right) e^{ikq\Delta x}$$

Numerical wave propagation (1)

Inserting a basic wave component $u = e^{i(kx_q - \tilde{\omega} t_n)}$ in the scheme (64) requires computation of

$$
\begin{aligned}
\lbrack D_tD_t e^{ikx}e^{-i\tilde{\omega} t}\rbrack^n_q &= \lbrack D_tD_t e^{-i\tilde{\omega} t}\rbrack^n e^{ikq\Delta x}\\
&= -\frac{4}{\Delta t^2}\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) e^{-i\tilde{\omega} n\Delta t}e^{ikq\Delta x} \qquad (65)\\
\lbrack D_xD_x e^{ikx}e^{-i\tilde{\omega} t}\rbrack^n_q &= \lbrack D_xD_x e^{ikx}\rbrack_q e^{-i\tilde{\omega} n\Delta t}\\
&= -\frac{4}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right) e^{ikq\Delta x}e^{-i\tilde{\omega} n\Delta t} \qquad (66)
\end{aligned}
$$

Numerical wave propagation (2)

The complete scheme,

$$
\lbrack D_tD_t e^{ikx}e^{-i\tilde{\omega} t} = c^2 D_xD_x e^{ikx}e^{-i\tilde{\omega} t}\rbrack^n_q
$$

leads to an equation for $\tilde{\omega}$:

$$
\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C^2\sin^2\left(\frac{k\Delta x}{2}\right), \qquad (67)
$$

where $C = \frac{c\Delta t}{\Delta x}$ is the Courant number

Numerical wave propagation (3)

Taking the square root of (67):

$$
\sin\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C\sin\left(\frac{k\Delta x}{2}\right), \qquad (68)
$$

- Exact ω is real
- Look for a real solution $\tilde{\omega}$ of (68)
- Then the sine functions are in $[-1, 1]$
- Lef-hand side in $[-1, 1]$ requires $C \leq 1$

Stability criterion

$$
C = \frac{c\Delta t}{\Delta x} \leq 1 \qquad (69)
$$

Why $C \leq 1$ is a stability criterion

Assume $C > 1$. Then

$$
\underbrace{\sin\left(\frac{\tilde{\omega}\Delta t}{2}\right)} > 1 = C\sin\left(\frac{k\Delta x}{2}\right)
$$

- $|\sin x| > 1$ implies complex x
- Here: complex $\tilde{\omega} = \tilde{\omega}_r \pm i\tilde{\omega}_i$
- One $\tilde{\omega}_i < 0$ gives $\exp(i\cdot i\tilde{\omega}_i) = \exp(\tilde{\omega}_i)$ and exponential growth

Numerical dispersion relation

- How close is $\tilde{\omega}$ to ω?
- Can solve for an explicit formula for $\tilde{\omega}$

$$
\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(C\sin\left(\frac{k\Delta x}{2}\right)\right) \qquad (70)
$$

- $\omega = kc$ is the *analytical dispersion relation*
- $\tilde{\omega} = \tilde{\omega}(k, c, \Delta x, \Delta t)$ is the *numerical dispersion relation*
- Speed of waves: $c = \omega/k$, $\tilde{c} = \tilde{\omega}/k$
- The numerical wave component has a wrong, mesh-dependent speed

The special case $C = 1$

- For $C = 1$, $\tilde{\omega} = \omega$
- The numerical solution is exact (at the mesh points)!
- The only requirement is constant c

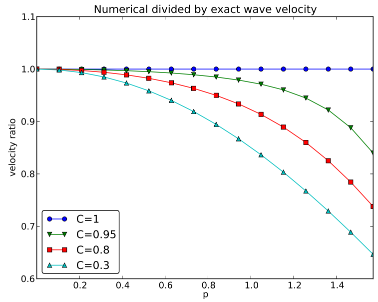
Computing the error in wave velocity

- Introduce $p = k\Delta x/2$
- p measures no of mesh points in space per wave length in space
- Study error in wave velocity through $\tilde{c}/c$ as function of p

$$r(C,p) = \frac{\tilde{c}}{c} = \frac{1}{Cp}\sin^{-1}(C\sin p), \quad C \in (0,1],\ p \in (0,\pi/2]$$

Visualizing the error in wave velocity

```
def r(C, p):
    return 2/(C*p)*asin(C*sin(p))
```



Note: the shortest waves have the largest error, and short waves

Taylor expanding the error in wave velocity

For small p, Taylor expand $\tilde{\omega}$ as polynomial in p:

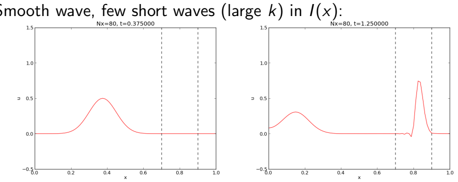
```
>>> C, p = symbols('C p')
>>> rs = r(C, p).series(p, 0, 7)
>>> print rs
1 - p**2/6 + p**4/120 - p**6/5040 + C**2*p**2/6 -
C**2*p**4/12 + 13*C**2*p**6/720 + 3*C**4*p**4/40 -
C**4*p**6/16 + 5*C**6*p**6/112 + O(p**7)

>>> # Factorize each term and drop the remainder O(...) term
>>> rs_factored = [factor(term) for term in rs.lseries(p)]
>>> rs_factored = sum(rs_factored)
>>> print rs_factored
p**6*(C - 1)*(C + 1)*(225*C**4 - 90*C**2 + 1)/5040 +
p**4*(C - 1)*(C + 1)*(3*C - 1)*(3*C + 1)/120 +
p**2*(C - 1)*(C + 1)/6 + 1
```

Leading error term is $\frac{1}{6}(C^2-1)p^2$ or

$$\frac{1}{6}\left(\frac{k\Delta x}{2}\right)^2(C^2-1) = \frac{k^2}{24}\left(c^2\Delta t^2 - \Delta x^2\right) = \mathcal{O}(\Delta t^2, \Delta x^2) \quad (71)$$

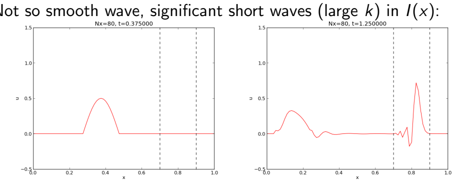
Example on effect of wrong wave velocity (1)

Smooth wave, few short waves (large k) in $I(x)$:



Example on effect of wrong wave velocity (1)

Not so smooth wave, significant short waves (large k) in $I(x)$:



Extending the analysis to 2D (and 3D)

$$u(x,y,t) = g(k_x x + k_y y - \omega t)$$

is a typically solution of

$$u_{tt} = c^2(u_{xx} + u_{yy})$$

Can build solutions by adding complex Fourier components of the form

$$e^{i(k_x x + k_y y - \omega t)}$$

Discrete wave components in 2D

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u)]^n_{q,r} \tag{72}$$

This equation admits a Fourier component

$$u^n_{q,r} = e^{i(k_x q\Delta x + k_y r\Delta y - \tilde{\omega} n\Delta t)} \tag{73}$$

Inserting the expression and using formulas from the 1D analysis:

$$\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C_x^2 \sin^2 p_x + C_y^2 \sin^2 p_y, \tag{74}$$

where

$$C_x = \frac{c^2\Delta t^2}{\Delta x^2},\quad C_y = \frac{c^2\Delta t^2}{\Delta y^2},\quad p_x = \frac{k_x\Delta x}{2},\quad p_y = \frac{k_y\Delta y}{2}$$

Stability criterion in 2D

Rreal-valued $\tilde{\omega}$ requires

$$C_x^2 + C_y^2 \leq 1 \tag{75}$$

or

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-1/2} \tag{76}$$

Stability criterion in 3D

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)^{-1/2} \tag{77}$$

For $c^2 = c^2(\mathbf{x})$ we must use the worst-case value $\bar{c} = \sqrt{\max_{\mathbf{x}\in\Omega} c^2(\mathbf{x})}$ and a safety factor $\beta \leq 1$:

$$\Delta t \leq \beta\frac{1}{\bar{c}}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)^{-1/2} \tag{78}$$

Numerical dispersion relation in 2D (1)

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\left(C_x^2 \sin^2 p_x + C_y^2 \sin^p_y\right)^{\frac{1}{2}}\right)$$

For visualization, introduce θ:

$$k_x = k\sin\theta,\quad k_y = k\cos\theta,\quad p_x = \frac{1}{2}kh\cos\theta,\quad p_y = \frac{1}{2}kh\sin\theta$$

Also: $\Delta x = \Delta y = h$. Then $C_x = C_y = c\Delta t/h \equiv C$.

Now $\tilde{\omega}$ depends on

- C reflecting the number cells a wave is displaced during a time step
- kh reflecting the number of cells per wave length in space
- θ expressing the direction of the wave

Numerical dispersion relation in 2D (2)

$$\frac{\tilde{c}}{c} = \frac{1}{Ckh}\sin^{-1}\left(C\left(\sin^2\left(\frac{1}{2}kh\cos\theta\right) + \sin^2\left(\frac{1}{2}kh\sin\theta\right)\right)^{\frac{1}{2}}\right)$$

Can make color contour plots of $1 - \tilde{c}/c$ in *polar coordinates* with θ as the angular coordinate and kh as the radial coordinate.

Numerical dispersion relation in 2D (3)